4.1 Tabular Data

We've seen how Python can store collections of data, such as lists, sets, and dictionaries. Mostly, we've focused on collections of integers or strings. But what about collections of collections? We've actually encountered this already: our `count_cancelled` function had a parameter `flights` that was a dictionary whose values were lists, and we represented the *Loves* predicate as a list of lists, storing a two-dimensional table of booleans. In this section, we'll look at using list of lists to store more complex forms of tabular data, like a table from a spreadsheet, and writing functions to perform computations on this data.

Toronto getting married

Let's consider a real data set from the city of Toronto. This data shows information about how many marriage licenses were issued in Toronto at a particular location and month. The data is in a tabular format with four columns: id, civic centre, number of marriage licenses issued, and time period. Each row of the table tells us how many marriage licenses were issued by a civic centre in a specific time period; the id is simply a unique numerical identifier for each row. Suppose we wanted to answer the following question: What is the average number of marriage licenses issued by each civic centre?

ID	**Civic Centre**	**Marriage Licenses Issued**	**Time Period**
1657	ET	80	January 1, 2011
1658	NY	136	January 1, 2011
1659	SC	159	January 1, 2011
1660	TO	367	January 1, 2011
1661	ET	109	February 1, 2011
1662	NY	150	February 1, 2011
1663	SC	154	February 1, 2011
1664	TO	383	February 1, 2011

To write a program that uses this data, we must first decide on a way to store it. As we did with our *Loves* table of values, we'll store this table as a list of lists, where each inner list represents one row of the table. Unlike our previous example, these lists won't just store boolean values, so we need to determine what data type to use for each column, based on the sample data we have.

- The ids and number of marriage licenses are natural numbers, so we'll use the `int` data type for them.
- The civic centre is a two-letter code, and so we'll store it as a `str`.

- The time period is a year-month combination; we'll represent these as dates using the `datetime` module.[1]

[1] To review this `date` data type, check out 2.4 Importing Modules.

With this in mind, let us see how we can store our data as a nested list[2]:

[2] In tutorial, you will explore how to load the data from a file into a nested list.

```
>>> import datetime
>>> marriage_data = [
...     [1657, 'ET', 80, datetime.date(2011, 1, 1)],
...     [1658, 'NY', 136, datetime.date(2011, 1, 1)],
...     [1659, 'SC', 159, datetime.date(2011, 1, 1)],
...     [1660, 'TO', 367, datetime.date(2011, 1, 1)],
...     [1661, 'ET', 109, datetime.date(2011, 2, 1)],
...     [1662, 'NY', 150, datetime.date(2011, 2, 1)],
...     [1663, 'SC', 154, datetime.date(2011, 2, 1)],
...     [1664, 'TO', 383, datetime.date(2011, 2, 1)]
... ]
>>> len(marriage_data)  # There are eight rows of data
8
>>> len(marriage_data[0])  # The first row has four elements
4
>>> [len(row) for row in marriage_data]  # Every row has four elements
[4, 4, 4, 4, 4, 4, 4, 4]
>>> marriage_data[0]
[1657, 'ET', 80, datetime.date(2011, 1, 1)]
>>> marriage_data[1]
[1658, 'NY', 136, datetime.date(2011, 1, 1)]
```

We can see that by indexing the nested list `marriage_data`, a list is returned. Specifically, this list represents a row from our table. For each row, we can then access its id via index 0, its civic centre via index 1, and so on.

```
>>> marriage_data[0][0]
1657
>>> marriage_data[0][1]
'ET'
>>> marriage_data[0][2]
80
>>> marriage_data[0][3]
datetime.date(2011, 1, 1)
```

Accessing columns and filtering rows

Suppose we want to see all of the different values from a single column of this table (e.g., all civic centres or marriage license numbers). We can retrieve a column by using a list comprehension:

```
>>> [row[1] for row in marriage_data]  # The civic centre column
['ET', 'NY', 'SC', 'TO', 'ET', 'NY', 'SC', 'TO']
```

Or, using an identically-structured set comprehension, we can obtain all unique values in a column.

```
>>> {row[1] for row in marriage_data}
{'NY', 'TO', 'ET', 'SC'}
```

Using our knowledge of filtering using if conditions in comprehensions, we can retrieve all rows corresponding to a specific civic centre.

```
>>> [row for row in marriage_data if row[1] == 'TO']
[[1660, 'TO', 367, datetime.date(2011, 1, 1)], [1664, 'TO', 383,
         datetime.date(2011, 2, 1)]]
```

Or we can filter rows based on a threshold for the number of marriage licenses issued:

```
>>> [row for row in marriage_data if row[2] > 380]
[[1664, 'TO', 383, datetime.date(2011, 2, 1)]]
```

A worked example

Earlier, we asked the question: What is the average number of marriage licenses issued by each civic centre? The question implies a mapping of civic centre names to numbers (i.e., the average). This means we need to create a dictionary comprehension. Let's start exploring in the Python console. Remember, we saw earlier that we can get all unique civic centre names in the data through a set comprehension.

```
>>> names = {row[1] for row in marriage_data}
>>> names
{'NY', 'TO', 'ET', 'SC'}
>>> {key: 0 for key in names}
{'NY': 0, 'TO': 0, 'ET': 0, 'SC': 0}
```

So far, we've created a dictionary where each key is a civic centre name and they all map to the value 0. To proceed, we need to be able to calculate the average number of marriage licenses issued per month by each civic centre.

Let's try to do this just for the `'TO'` civic centre first. We saw earlier how to get all rows for a specific civic centre, and to extract the values for a specific column. We'll first combine these two operations to retrieve the number of marriage licenses issued by `'TO'` each month.

```
>>> [row for row in marriage_data if row[1] == 'TO']  # The 'TO' rows
[[1660, 'TO', 367, datetime.date(2011, 1, 1)], [1664, 'TO', 383,
         datetime.date(2011, 2, 1)]]
>>> [row[2] for row in marriage_data if row[1] == 'TO']  # The 'TO' marriages
         issued
[367, 383]
>>> issued_by_TO = [row[2] for row in marriage_data if row[1] == 'TO']
```

So `issued_by_TO` is now a list containing the number of marriage licenses issued by the `'TO'` civic centre. We can now calculate their average by dividing its sum by its length:

```
>>> sum(issued_by_TO) / len(issued_by_TO)
375.0
```

Excellent! Through our exploration, we managed to find the average number of marriage licenses issued by one specific civic centre. How can we merge this with our earlier dictionary comprehension? It's quite a bit to keep in our head at once, and looks like it will quickly get messy. At this point, we should design a function to help us. Specifically, let's design a function that calculates the average for only one civic centre. As input, we will need the dataset as well as the name of the civic centre we are querying.

```
def average_licenses_issued(data: list[list], civic_centre: str) -> float:
    """Return the average number of marriage licenses issued by civic_centre
         in data.

    Return 0.0 if civic_centre does not appear in the given data.

    Preconditions:
      - all({len(row) == 4 for row in data})
      - data is in the format described in Section 4.1
    """
    issued_by_civic_centre = [row[2] for row in data if row[1] ==
         civic_centre]

    if issued_by_civic_centre == []:
        return 0.0
    else:
        total = sum(issued_by_civic_centre)
        count = len(issued_by_civic_centre)

        return total / count
```

Let’s test it to make sure we get the same result as before:

```
>>> average_licenses_issued(marriage_data, 'TO')
375.0
```

Finally, we can combine it with our previous dictionary comprehension by observing that `'TO'` can be replaced with the `key` that is changing:

```
>>> {key: 0 for key in names}
{'NY': 0, 'TO': 0, 'ET': 0, 'SC': 0}
>>> {key: average_licenses_issued(marriage_data, key) for key in names}
{'NY': 143.0, 'TO': 375.0, 'ET': 94.5, 'SC': 156.5}
```

Now that we’ve done this exploration in the Python console, we can save our work by writing this as a function:

```
def average_licenses_by_centre(marriage_data: list[list]) -> Dict[str,
        float]:
    """Return a mapping of the average number of marriage licenses issued at
        each civic centre.

    In the returned mapping:
      - Each key is the name of a civic centre
      - Each corresponding value is the average number of marriage licenses
        issued at
      that centre.

    Preconditions:
      - marriage_data is in the format described in Section 4.1
    """
    names = {'TO', 'NY', 'ET', 'SC'}
    return {key: average_licenses_issued(marriage_data, key) for key in
        names}
```